Voting Power

Assignment written by Julie Zelenski

Block voting systems

It's said that "every vote counts," but does every vote count equally? A block voting system such as the U.S. Electoral College makes for an interesting case study in understanding the relative voting power of a system of variably-sized block votes.

In a block voting system, each block has an assigned number of votes, and the votes for one block are cast in unison. In the U.S. electoral system, California has a block of 55 votes while New Mexico has 5. Does this mean that California wields 10 times the influence of New Mexico in affecting the election outcome? Let's explore further!

Define a *coalition* to be a group of blocks that all vote for the same candidate. A set of **n** voting blocks can form **2^n** distinct coalitions. Some of those coalitions would win the election, others would lose. One measure of a block's importance or voting "power" is the percentage of coalitions in which that block's vote is critical to winning the election. A **critical** or **swing** vote is one that changes the election outcome, i.e. including this block in the coalition tips the scales, changing what was a losing coalition into a winner. The count of coalitions for which a block has a critical vote is used to compute its **Banzhaf Power Index**, a measure of the voting power wielded by the block.

Banzhaf Power Index

For a given voting block B, we count the coalitions in which B has a critical vote by identifying winning coalitions in which B can participate but in which that coalition would not win if B does not join. B joining the coalition supplies a critical vote that swings the election outcome.

Consider this example system of three blocks:

Block ID	Block Count
Lions	50
Tigers	49
Bears	1

First let's enumerate the eight possible coalitions that could form: **`Lions+Tigers+Bears`**, **`Lions+Tigers`**, **`Lions+Bears`**, **`Tigers+Bears`**, **`Lions`**, **`Tigers`**, **`Bears`**, **`{}`** (the empty coalition).

Winning an election requires a strict majority (i.e. more than half of the total votes). In this example, there are 100 total votes, thus to win the election, a coalition must amass 51 or more votes.

Of the eight possible coalitions, three are winning coalitions: **L+T+B** (100 votes), **L+T** (99 votes), and **L+B** (51 votes).

For each winning coalition, consider which of its blocks contribute a critical vote:

- **`Lions`** has a critical vote in **L+T+B**, **L+T**, and **L+B**
- **`Tigers`** has a critical vote in **L+T**
- **`Bears`** has a critical vote in **L+B**

A block supplies a critical vote if its support for the coalition changes the election outcome. A winning coalition would no longer be winning if a critical voter left the coalition. Note that **`Lions`** is a critical vote for the coalition **L+T+B**, but neither **`Tigers`** nor **`Bears`** is.

Counting the critical votes for each block gives us the following data:

Block ID	Critical Votes
Lions	3
Tigers	1
Bears	1

The **Banzhaf Power Index** expresses a block's voting power as the percentage of critical votes that this block has out of all total critical votes in the system. To convert from the count of critical votes to the Power Index, sum all critical votes in the system and compute the percentage per block. For example, this system has 5 total critical votes of which **`Lions`** have 3, so **`Lions`** control 3/5 or 60% of the critical votes. The table below shows the power indexes for the example system:

Block ID	Banzhaf Power Index
Lions	60% (= 3/5)
Tigers	20% (= 1/5)
Bears	20% (= 1/5)

Comparing relative percentages shows the difference in voting power among the blocks.

`Tigers` and **`Bears`** have equivalent voting power, despite the **`Tigers`**' much larger block count. The small uptick in block count for the **`Lions`** gives it three times more voting power than the **`Tigers`**. Apparently, the lion's share of the votes really does go the **`Lions`**! (sorry… we could not resist)

Your task

You are to write the function

```
Map<string,int> computePowerIndexes(Vector<blockT>& blocks)
```

which receives a Vector of size N that contains all of the blocks in the system. A **`blockT`** is a struct containing two fields, the block's id string and its block count.

```
struct blockT {
    string id;
    int count;
};
```

The function returns a Map of size N which associates each block id with the Banzhaf power index for that block. Calling **`computePowerIndexes`** on the vector of blockTs **`{ {"Lions",50}, {"Tigers",49}, {"Bears",1} }`** returns a map with the entries shown in the table above (i.e. **`map["Lions"] = 60`**, and so on for each block id).

Although the explanation above has described the process in terms of enumerating the coalitions and then determining which participating blocks are critical to each, it is more straightforward to structure the code to count the critical votes specific to a target block and repeat that process for all blocks. Here is our suggestion:

- Choose a target block for which you want to count its critical votes.
  - Set aside the target block and recursively explore the coalitions that can be formed from the remaining blocks, not including the target. As you form a coalition, do not store the collection of participating blocks, just track the amassed vote total.
  - Once a coalition is formed and you know its vote total, now consider the impact of the target block's participation. Without the target block, does the coalition lose, and with the target block, does the coalition win? If the answer to **both** of these questions is yes, then the target block is a critical vote for this coalition.
  - Repeat the above two steps to explore all possible coalitions and tally those for which target is a critical vote.
- Now choose a different block as the target and repeat the process. Continue until all blocks have had a turn as target.
- After having tallied all critical votes, the conversion to percentage per block is simple looping and arithmetic.

After testing and debugging your function, predict what you expect to be the Big O of the `computePowerIndex` function. Then use the timing operation to measure the execution time over 5 or more different sizes chosen to confirm your prediction. Choose sizes to fit your hardware — big enough to be measurable/stable but not so long that you fall asleep waiting. Try to set your largest operation so that it completes in under a minute or so.

Answer these questions in `short_answer.txt`:

Q7. What is the Big O of `computePowerIndex`? Include your timing data and explain how it supports your reasoning.

Q8. Use the Big O and timing data to estimate how long it would take to compute the power index for the 51 voting blocks in the U.S. Electoral College.

Notes

- **Ties**. To win the election, a coalition must earn a strict majority of the votes. A tie is not considered a winning coalition.
- **Rounding**. We calculate the power index as an integer which is a rounded value (well, to be more correct about what happens, the decimal portion is *truncated*). The sum over all blocks may be less than the total 100% because of this truncation. For example, three blocks each holding a third of voting power (33.333 %) would truncate to power indexes of 33 each and sum to 99. This is expected behavior and nothing to worry about.
- The recursive insight is the **classic include/exclude pattern** used for subset exploration. Starting with similar code you've seen in lecture/section/warmup will be a good start, but you'll need to modify some of the details to add the housekeeping and counting required for this problem.
- **Efficiency**. The exhaustive recursion to try all subsets is computationally expensive. Here are a few things to tame its resource-hungry nature:
    - As soon as it is apparent that a coalition is going to win regardless of whether or not the target block participates, there is no need to further explore that path. The target block cannot be a critical vote in this coalition.
    - Be thoughtful about use of ADTs and take care to avoid unneeded copy operations (expensive!). A data structure is copied when passed by value (not reference) or returned from a function. A few copies here or there is no problem, but making a copy on every single recursive call of the entire exploration? No bueno!
    - Furthermore, do **not** attempt to first construct the full power set of all coalitions (subsets) and then process the coalitions. Given a voting system with even a modest number of blocks, this would consume a prohibitive amount of memory. Instead you must explore the coalitions one at a time: assemble a coalition, test it, and then un-choose and backtrack to consider other choices in forming another coalition. At any one time, there is only one coalition being formed/stored.
    - Counting critical votes will make a **lot** of recursive calls, which means it is extra important that within each recursive call that you do not do expensive operations such as make a copy of the input `Vector` or edit the `Vector` contents by removing and inserting elements! Given a larger input `Vector`, these expensive operations will slow your function to an absolute crawl. One technique shown in section/lecture examples was to preserve the one original `Vector` as-is and use an index to track your progress through the recursive calls. Review the code for subset sum from the warmup as one example. This makes a huge difference in efficiency!
    - As a point of reference, our (straightforward, no-tricks) solution running on a vector of 24 blocks completes in about 5 seconds on a modestly-equipped Mac. If your code is taking much much longer, it could indicate you have something amiss. Post to Ed or bring it to the Lair and we can help.
- **Duplicates/re-calculating**. Two or more blocks in a system can have the same number of votes; the number of critical votes for blocks with equal numbers of votes will be the same. You do not have to do something clever to avoid this repetition; you may compute the number of critical votes for each block anew. If you are eager to be clever about it, consider tackling the extension.

- **Helper/wrapper functions**. There is a firm requirement that the function `computePowerIndexes` is implemented to exactly match the prototype above. You are free to decide the names and parameters of any additional helper/wrapper functions you use. You will almost certainly need at least one helper function.

References

- The Banzhaf Power Index originated in a lawsuit raised by John Banzhaf challenging the fairness of a block-voting system used in Nassau County, New York. He argued that the system with block counts of `{9,9,7,3,1,1}` disenfranchised the three smallest blocks, as they never cast a critical vote and had zero voting power! Wikipedia: Banzhaf Power Index
- Banzhaf making recent news as applied to analyzing blockchain voting systems.

Extension

Exponential problems can be a tough nut to efficiently crack. The approach we are using to solve the problem repeats many calculations, which further bogs it down. Research how you could apply techniques such as memoization, dynamic programming, or generating functions to achieve a more efficient implementation.